# [谜之RxJava（四）—— Retrofit和RxJava的基情](https://segmentfault.com/a/1190000004077117)

我们今天着重来介绍下RxJavaCallAdapterFactory这个类。用过的朋友们都知道，它是用来把Retrofit转成RxJava可用的适配类。

## RxJavaCallAdapterFactory

OK，首先看下声明

public final class RxJavaCallAdapterFactory implements CallAdapter.Factory

CallAdapter.Factory是Retrofit这个库中的接口，用来给我们自定义去解析我们自己想要的类型用的。

举个例子：

@GET("/aaa")  
Observable<QuestionListData> getQuestionNewestList();

这么一个接口，retrofit本身是无法识别Observable<QuestionListData>然后去工作的，如果没有这个适配器就根本无法工作，因此我们的适配器的作用，就是生成我们想要的Observable。

看下它的实现。

@Override  
public CallAdapter<?> get(Type returnType, Annotation[] annotations, Retrofit retrofit) {  
 Class<?> rawType = Utils.getRawType(returnType);  
 boolean isSingle = "rx.Single".equals(rawType.getCanonicalName());  
 if (rawType != Observable.class && !isSingle) {  
 return null;  
 }  
 if (!(returnType instanceof ParameterizedType)) {  
 String name = isSingle ? "Single" : "Observable";  
 throw new IllegalStateException(name + " return type must be parameterized"  
 + " as " + name + "<Foo> or " + name + "<? extends Foo>");  
 }  
  
 CallAdapter<Observable<?>> callAdapter = getCallAdapter(returnType);  
 if (isSingle) {  
 // Add Single-converter wrapper from a separate class.   
 // This defers classloading such that  
 // regular Observable operation can be leveraged without   
 // relying on this unstable RxJava API.  
 return SingleHelper.makeSingle(callAdapter);  
 }  
 return callAdapter;  
}

这里代码的意思就是说，如果你返回的不是Observable<T>这种类型，我就不干！  
如果是的话，那我再来详细看下模板类是哪个，也就是getCallAdapter接口干的事。

private CallAdapter<Observable<?>> getCallAdapter(Type returnType) {  
 Type observableType = Utils.getSingleParameterUpperBound((ParameterizedType) returnType);  
 Class<?> rawObservableType = Utils.getRawType(observableType);  
 if (rawObservableType == Response.class) {  
 if (!(observableType instanceof ParameterizedType)) {  
 throw new IllegalStateException("Response must be parameterized"  
 + " as Response<Foo> or Response<? extends Foo>");  
 }  
 Type responseType = Utils.getSingleParameterUpperBound((ParameterizedType) observableType);  
 return new ResponseCallAdapter(responseType);  
 }  
  
 if (rawObservableType == Result.class) {  
 if (!(observableType instanceof ParameterizedType)) {  
 throw new IllegalStateException("Result must be parameterized"  
 + " as Result<Foo> or Result<? extends Foo>");  
 }  
 Type responseType = Utils.getSingleParameterUpperBound((ParameterizedType) observableType);  
 return new ResultCallAdapter(responseType);  
 }  
  
 return new SimpleCallAdapter(observableType);  
}

这里告诉我们，除了Observable<Response>和Observable<Result>需要不同的Adapter处理外，其他的都让SimpleCallAdapter处理。

OK，我们就不看别的，直捣黄龙，看SimpleCallAdapter！

## SimpleCallAdapter 创建Observable的类

static final class SimpleCallAdapter implements CallAdapter<Observable<?>> {  
 private final Type responseType;  
  
 SimpleCallAdapter(Type responseType) {  
 this.responseType = responseType;  
 }  
  
 @Override public Type responseType() {  
 return responseType;  
 }  
  
 @Override public <R> Observable<R> adapt(Call<R> call) {  
 return Observable.*create*(new CallOnSubscribe<>(call)) //  
 .flatMap(new Func1<Response<R>, Observable<R>>() {  
 @Override public Observable<R> call(Response<R> response) {  
 if (response.isSuccess()) {  
 return Observable.*just*(response.body());  
 }  
 return Observable.*error*(new HttpException(response));  
 }  
 });  
 }  
}

这里总算看见我们熟悉的Observable接口咯，原来是自己定义了一个OnSubscribe，然后把Response通过flatMap转为我们想要的对象了。 这里同时也判断请求是否成功，进入Observable的工作流里了。

好，我们最终可以看下CallOnSubscribe干了啥

这static final class CallOnSubscribe<T> implements Observable.OnSubscribe<Response<T>> {  
 private final Call<T> originalCall;  
  
 private CallOnSubscribe(Call<T> originalCall) {  
 this.originalCall = originalCall;  
 }  
  
 @Override public void call(final Subscriber<? super Response<T>> subscriber) {  
 // Since Call is a one-shot type, clone it for each new subscriber.  
 final Call<T> call = originalCall.clone();  
  
 // Attempt to cancel the call if it is still in-flight on unsubscription.  
 subscriber.add(Subscriptions.*create*(new Action0() {  
 @Override public void call() {  
 call.cancel();  
 }  
 }));  
  
 if (subscriber.isUnsubscribed()) {  
 return;  
 }  
  
 try {  
 Response<T> response = call.execute();  
 if (!subscriber.isUnsubscribed()) {  
 subscriber.onNext(response);  
 }  
 } catch (Throwable t) {  
 Exceptions.*throwIfFatal*(t);  
 if (!subscriber.isUnsubscribed()) {  
 subscriber.onError(t);  
 }  
 return;  
 }  
  
 if (!subscriber.isUnsubscribed()) {  
 subscriber.onCompleted();  
 }  
 }  
}

里其实蛮简单的，call是retrofit对okhttp的一个代理，是一个同步网络请求，在这里就是典型的对网络进行数据请求，完了放到subscriber的onNext里，完成网络请求。我们可以看下，它把unsubscribe，也就是取消请求的情况处理的挺好。

subscriber.add(Subscriptions.create(new Action0() {  
 @Override

public void call() {  
 call.cancel();  
 }  
}));

这段代码是给subscribe增加一个unsubscribe的事件。 也就是请求完成的时候，会自动对call进行一个终止，也就是http的close行为。

## 前方高能

今天被坑到这里很久，我们对API调用了observeOn(MainThread)之后，线程会跑在主线程上，包括onComplete也是，unsubscribe也在主线程，然后如果这时候调用call.cancel会导致NetworkOnMainThreadException，所以一定要在retrofit的API调用ExampleAPI.subscribeOn(io).observeOn(MainThread)之后加一句unsubscribeOn(io)。

完整的就是ExampleAPI.subscribeOn(io).observeOn(MainThread).unsubscribeOn(io)。